Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем

Системное программирование

Никольский Кирилл Андреевич

Формальная верификация окружения времени выполнения робототехнической системы

Бакалаврская работа

Научный руководитель:
д. ф.-м. н., профессор Терехов А. Н.

Рецензент:
к.т.н, доцент Ицыксон В. М.

Санкт-Петербург
2016

SAINT-PETERSBURG STATE UNIVERSITY

Software and Administration of Information Systems

Software Engineering

Nikolskiy Kirill

Formal verification of robotics runtime environment

Graduation Thesis

Scientific supervisor:
professor Andrey Terekhov

Reviewer:
assistant professor Vladimir Itsykson

Saint-Petersburg
2016

Оглавление

Введение

Компьютерные программы часто содержат различные ошибки. Еще Эдсгер Вибе Дейкстра говорил, что тестирование не позволяет обнаружить все ошибки. Особенно трудно тестировать параллельные, распределенные и многопоточные программы. Даже в тех случаях, когда функционирование каждой из взаимодействующих параллельных частей системы полностью ясно, человеку тяжело осознать работу параллельной системы целиком. Параллельные системы, на первый взгляд, работающие правильно, могут содержать ошибки, проявляющиеся в очень редких случаях (тупики, гонки [24] и т.д.). Существует множество примеров, когда из-за редко проявляющихся ошибок в сложных системах, прошедших тщательное тестирование, компании понесли огромные потери или пострадали пользователи[1] [21].

В современном мире имеется большое количество автоматических систем, например автопилот в самолете или спутник, и критически важна корректная и безотказная работа всех компонентов таких систем. Поэтому на первый план выходят свойства надежности и предсказуемости поведения системы. Таким образом, *формальная верификация* программного кода, позволяющая существенно повысить качество системы, становится важнейшей областью научных исследований в информатике. Формальной верификацией программы, согласно [21], будем называть приемы и методы формального доказательства (или опровержения) того, что модель программной системы удовлетворяет заданной формальной спецификации (см. рис. 1).

Уже сейчас верификация является промышленным стандартом в критических областях, связанных как с дорогостоящими проектами, так и с человеческими жизнями [4]. С помощью формальной верификации можно доказать некоторые важные свойства системы, гарантировав отсутствие ошибок определенного типа и правильное функционирование системы. Это происходит за счет того, что верификация позволяет провести исчерпывающую проверку всех возможных вычислений

[1]URL: http://catless.ncl.ac.uk/Risks

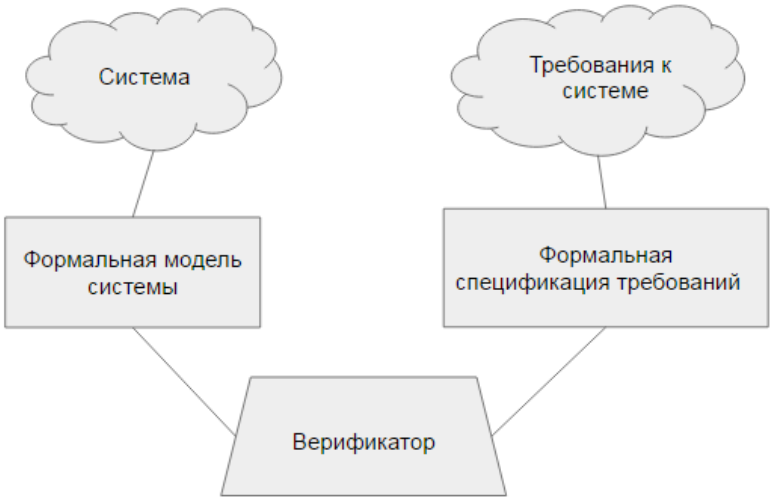

Рис. 1: Общая схема верификации

модели системы, обнаруживая редко проявляющиеся ошибки [20], которые очень сложно проверить тестированием. Важно понимать, что проведя формальную верификацию и тестирование, нельзя гарантировать полное отсутствие ошибок в системе. Но при использовании этих методов вместе существенно повышается уровень доверия к системе.

На сегодняшний день для верификации сложных параллельных, многопоточных систем зарекомендовал себя метод проверки модели (model checking) [21]. Это метод проверки того, что данная формальная модель системы удовлетворяет ограничению, поставленному чаще всего в терминах какой-либо *темпоральной логики*, то есть логики, в которой истинность логических утверждений зависит от времени.

Важной частью автоматических систем являются роботы. Программное обеспечение для них часто относится к классу так называемых реагирующих систем, то есть систем, основная функция которых является поддержанием взаимодействия с окружением, а не преобразование информации. Такие системы могут бесконечно работать, они контролируют окружение, реагируя на внешние события. Вычисления на таких системах задаются бесконечной последовательностью состояний во времени. И именно с помощью формальной верификации возможно проанализировать все возможные вычисления таких систем.

В рамках Межвузовской Проектной Лаборатории Робототехники со-

здают программную часть для контроллера робототехнического конструктора TRIK [13]. На таком контроллере, помимо операционной системы, запущено *окружение времени выполнения* [16], предоставляющее интерфейс программирования приложений для программирования робота, используя язык C++ с инструментарием Qt[2] или скриптовый язык Qt Script[3], позволяющее исполнять на роботе скрипты, а также управлять роботом извне. Скриптовый язык упрощает программирование роботов, однако окружение времени выполнения является многопоточной, параллельной программой, что обуславливает высокую сложность обнаружения ошибок. Такие роботы могут использоваться как автоматические системы. И безошибочное функционирование окружения времени выполнения в таких случаях является важной задачей. Отметим, что сама программа, окружение времени выполнения, написана на языке C++ с помощью инструментария Qt.

Таким образом, в нашем случае наиболее подходящим методом для формальной верификации окружения времени выполнения робототехнической системы является метод проверки модели. В классическом виде наиболее трудоемкой частью данного метода является разработка самой модели окружения времени выполнения [21]. Сложность связана с большим размером программы и постоянно возникающей при моделировании проблемой комбинаторного взрыва [23]. Необходимо также описать в терминах темпоральной логики интересующие свойства данной программы. Остальная часть работы, связанная с проверкой свойств на модели, выполняется автоматизированно с помощью соответствующих инструментов верификации.

[2]Qt — кроссплатформенный инструментарий разработки ПО на языке программирования C++. URL: http://www.qt.io/

[3]QtScript — скриптовый язык, который является составной частью инструментария Qt, начиная с версии 4.3.0. Язык основан на стандарте ECMAScript с некоторыми расширениями, такими как возможность соединения с сигналами и слотами объектов QObject.

1. Цель и постановка задачи

Цель данной работы — формальная верификация окружения времени выполнения робототехнической системы.

В связи с этим были сформулированы следующие задачи.

- Сделать обзор существующих инструментов для формальной верификации программ по методу проверки модели.
- Создать корректную и адекватную модель окружения времени выполнения робототехнической системы.
- Сформулировать требования на естественном языке и написать спецификации к программе на основе ожидаемого поведения исходной системы.
- С помощью созданной модели окружения времени выполнения найти ошибки в оригинальной программе, которые могут проявиться при взаимодействии параллельных потоков.

Таким образом, результатами данной роботы должны стать модель окружения времени выполнения робототехнической системы и список зарегистрированных сообщений о найденных дефектах в системе отслеживания ошибок.

2. Обзор

В данном разделе описаны подходящие инструменты верификации и автоматической генерации модели. Сперва обосновывается в секции «Выбор метода верификации», почему вышеупомянутый метод проверки модели является наиболее подходящим методом для формальной верификации окружения времени выполнения. Затем формулируются требования к верификатору в рамках данной работы в секции «Требования к инструменту верификации», и в секции «Выбор верификатора» обосновывается выбор конкретного верификатора исходя из перечисленных в предыдущей секции требований. И в последней секции «Инструменты автоматической генерации модели» приводятся примеры соответствующих инструментов и обосновывается нецелесообразность их применения в данной работе.

2.1. Выбор метода верификации

В настоящее время в области формальной верификации можно выделить три основных направления, по которым активно проводятся исследования и разрабатываются соответствующие методы [21]:

- Дедуктивная верификация — это проверка правильности программы, которая сводится к доказательству теорем в подходящей логической системе. В ходе верификации вручную пишутся аннотации к коду, которые автоматически проверяются системами дедуктивной верификации, тем самым доказывается корректность работы каждой нити исполнения системы в отдельности. Однако на текущий момент отдельные потоки в исходной программе качественно протестированы и не имеют первоочередной важности для формальной верификации.

- Проверка эквивалентности — это проверка правильности программы, связанная с разработкой формальных моделей взаимодействующих процессов и проверкой эквивалентности поведений формальной модели системы и спецификации. Такой подход, в частно-

сти, удобен для верификации параллельной системы, написанной на языках, построенных на формализме исчисления взаимодействующих последовательных процессов (CSP) и, схожим с ним, языке исчисления взаимодействующих систем (CCS). Примерами таких языков являются occam, Go, Limbo. Однако код окружения времени выполнения написан на языке C++, что затрудняет его верификацию вышеописанным методом.

- Проверка модели (model checking) — это метод проверки того, что данная формальная модель системы удовлетворяет ограничению, поставленному чаще всего в терминах какой-либо темпоральной логики. Этот метод зарекомендовал себя для верификации сложных параллельных, многопоточных систем, включающих в себя взаимодействие с внешними компонентами. В нашем случае такой системой является окружение времени выполнения робототехнической системы. Стоит отметить, что этот метод может быть полностью автоматизирован при наличии формальной модели и формально заданных спецификаций.

Таким, образом мы показали, что метод проверки модели является наиболее подходящим для нахождения сложно воспроизводимых, многопоточных ошибок в исходной программе окружения времени выполнения.

2.2. Требования к инструменту верификации

В настоящее время существует большое количество инструментов верификации программ на моделях, среди которых присутствуют как коммерческие, так и свободно распространяемые с открытым исходным кодом. В рамках данной работы выбор будет осуществляться из последних. Для качественного решения поставленных задач необходимо выбрать инструмент верификации, позволяющий проверить различные требования (свойства) на моделях систем с параллельными взаимодействующими потоками. Соответственно, язык для моделирования таких

систем должен быть максимально удобным, позволяя быстро и понятно моделировать любые типы взаимодействий между потоками. Инструмент должен уметь выводить в результате проверки или сообщение об успехе, или контрпример (траекторию исполнения кода), на котором нарушается какое-либо свойство системы. Также важна поддержка режимов симуляции для ускорения и удобства разработки модели и анализа контрпримеров.

2.3. Выбор верификатора

В рамках данной работы были рассмотрены следующие свободно распространяемые инструменты верификации, являющиеся наиболее «зрелыми» для верификации параллельных систем:

- SPIN [2]
- UPPAAL [6]
- NuXMV [19]
- NuSMV [7]
- VIS [17]
- KRONOS [18]

В рамках работы [22], проведя сравнительный обзор, были рассмотрены четыре из них: UPPAAL, SPIN, NuSMV, NuXMV. В указанной работе требования к инструменту верификации отчасти совпадают с нашими и выбор в ней был сделан в пользу инструмента SPIN, поскольку этот инструмент поддерживает удобную синхронизацию процессов посредством рандеву-каналов, а также он имеет средство для моделирования асинхронного взаимодействия — асинхронные каналы с буфером. Такие возможности выгодно отличают выбранный инструмент SPIN от трех остальных.

В книге [12] приведен краткий обзор всех перечисленных инструментов, кроме инструмента VIS. В частности, было отмечено, что инструмент KRONOS обладает рядом недостатков, связанных как с отсутствием режима симуляции, так и с большим порогом вхождения. Наиболее подходящим для верификации параллельных систем и простым в освоении был признан инструмент верификации SPIN, несмотря на то, что он позволяет описывать лишь конечные системы.

В статье [8] был проведен сравнительных анализ двух инструментов верификации SPIN и VIS на примере верификации некоторого протокола методом проверки модели. Инструмент SPIN оказался наиболее подходящим для верификации в рамках данной работы по важным показателям, явно перечисленным в таблице в конце статьи.

Таким образом, из рассмотренных инструментов верификации для моделирования и проверки свойств окружения времени выполнения был выбран инструмент SPIN. Дополнительным аргументом в пользу выбора инструмента SPIN является большое количество материала и вводных курсов по этому верификатору на русском языке. Соответственно, моделирование системы будет происходить на входном для верификатора SPIN языке PROMELA[4]. Свойства моделируемой системы будут выражаться на языке LTL[5], а также с использованием *внутренних проверок* (*assert*)[6]. Стоит отметить, что в языке PROMELA отсутствуют встроенные возможности описывать системы с общими часами, в нем вообще нет понятия времени или часов. Для проверки свойств с временными рамками, что пока что не является приоритетной задачей, требуется выбрать другой инструмент верификации.

2.4. Инструменты автоматической генерации модели

Моделирование исходных программ без должного уровня абстракции не имеет смысла, так как в таком случае количество состояний в

[4]Process Meta Language, URL: http://spinroot.com/spin/Man/Intro.html

[5]Linear Temporal Logic [9]

[6]URL: http://spinroot.com/spin/Man/assert.html

модели будет огромным, а результатом запуска верификатора будет его остановка из-за нехватки памяти. На данный момент существует ряд исследований в области автоматической генерации модели из исходного кода [3, 1, 11]. Используя некоторые приемы и техники, не исключающие ручную работу, описанные в статьях инструменты генерации позволяют упростить создание модели в определенных случаях. Однако все такие инструменты не позволяют создавать модели (в частности, на языке PROMELA) для исходных систем больших масштабов, написанных на языке C++.

Таким образом, было признано целесообразным моделировать окружение времени выполнения вручную.

3. Модель системы

Исходная программа окружения времени выполнения содержит более 20000 строчек кода на языке C++. Для верификации была выбрана компонента окружения времени выполнения, которая, по эмпирическим наблюдениям, была наиболее подвержена сложно воспроизводимым ошибкам. Эта компонента отвечает за исполнение пользовательских скриптов (см. рис. 2). Содержание скриптов заранее неизвестно (то есть при моделировании необходимо учесть все возможные комбинации скриптовых вызовов функций из определенного набора), и, помимо прочего, они позволяют работать с потоками, создавая и завершая потоки в процессе исполнения скриптов, то есть поддерживается многопоточность исполнения. Подробная информация об архитектуре и процессе запуска скриптов приведена в [14] и [15].

Была построена документированная адекватная и корректная модель такой компоненты. Так как моделируемый код встречается в различных подсистемах проекта, сложно точно оценить объем смоделированного исходного кода, однако по примерным оценкам такой объем составляет примерно четверть проекта. Размер модели составляет чуть более 700 строчек кода на языке PROMELA, не считая кода, сгенерированного инструментом SPIN из LTL-формул. При этом в ходе верификации было исследовано порядка $4 \cdot 10^7$ состояний модели.

Для оценки корректности и адекватности модели были придуманы два приема. Один из них заключался в журналировании различных этапов работы системы и проверке соответствия получаемых сообщений. Поскольку в исходной программе на момент написания модели было реализовано журналирование, необходимо было лишь добавлять в процессе моделирования необходимую выводимую информацию в соответствующие места. Такая информация оказалась полезной на начальных этапах моделирования и при анализе контрпримеров.

Второй прием заключался в использовании динамического верификатора ThreadSanitizer[7], обнаруживающего гонки и потенциальные ту-

[7]URL: https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual

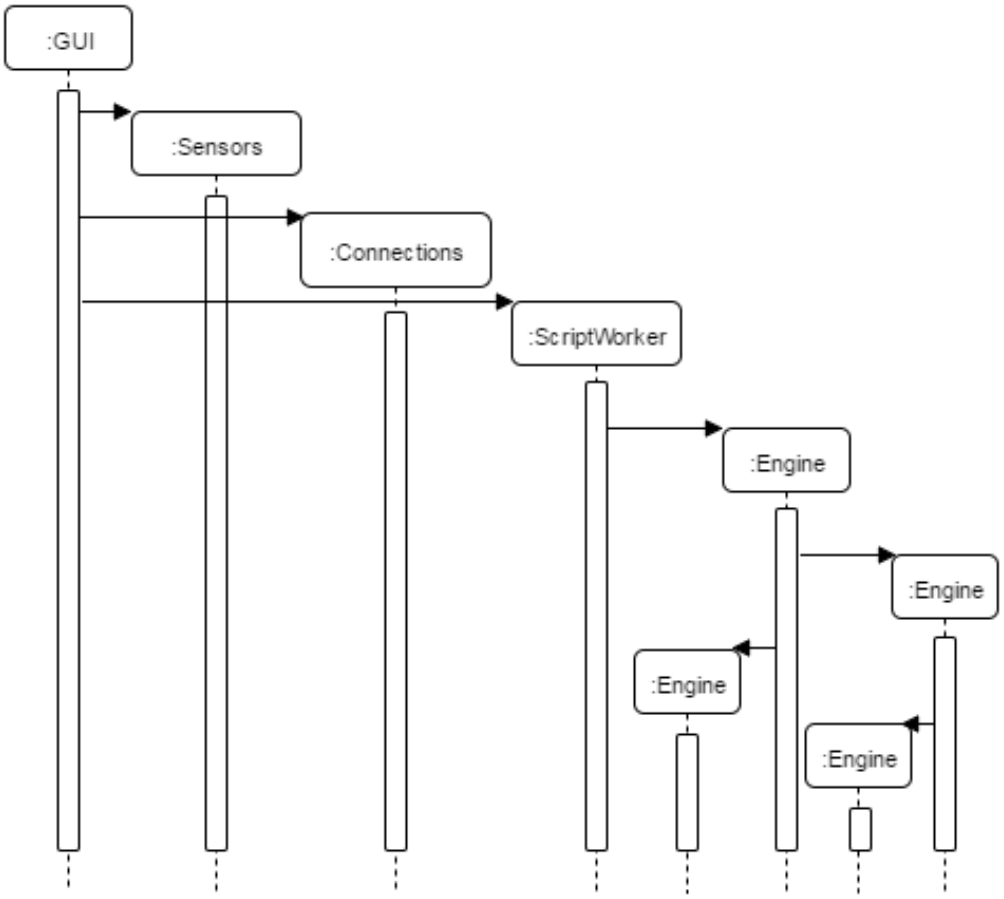

Рис. 2: Упрощенная диаграмма последовательности моделируемой системы, каждый из потоков Sensors и Connections на самом деле объединяет в себе группы потоков

пики в ходе работы программы. Выдаваемые таким инструментом сообщения (стеки вызовов в задействованных потоках) при обнаружении (потенциальных) ошибок, относящиеся к моделируемой системе, были использованы в качестве дополнительной информации для создания корректной и адекватной модели. Такие сообщения позволяли лучше понимать, от чего не стоит абстрагироваться, какие участки программы наиболее подвержены ошибкам.

Процесс создания модели разбивался на анализ исходной программы и моделирование работы отдельных компонентов и примитивов инструментария Qt:

- потоки,
- очереди событий,
- система сигналов-слотов,

- обработчики исключений,
- мьютексы.

Поток в нашей модели представляется процессом с бесконечным циклом обработки очереди событий, уникальной для каждого потока и являющейся асинхронным каналом с ограниченным буфером. Ограниченность буфера является проблемой, которая при верификации на данный момент решается установкой меток заключительного состояния (*end*)[8]. Иначе появляются множество ложных контрпримеров, связанных с невозможностью послать сообщение в заполненный канал, которые, очевидно, не воспроизводятся на исходной программе.

Система сигналов-слотов в зависимости от типа соединения (прямое или через очередь событий) моделируется или с помощью посылок сообщений в канал необходимого потока, или простым исполнением кода в текущем потоке. Таким образом, установленные соединения в исходной программе нужны лишь для определения их типов и нахождения потоков-получателей (за исключением некоторых проверок, связанных с местом объявления соединения). Все возможные сигналы описываются с помощью перечисляемого типа *mtype* и, так как заранее известны для каждого потока все соединения и сигналы, которые этот поток может обрабатывать, в нем и моделируется обработка *всех* возможных сигналов. Если необходимо передавать информацию, кроме названия сигнала, а это случается в нашей программе редко, то моделирование параметра происходит через глобальные переменные.

Обработчик исключения является в модели отдельным потоком, который может принимать управление и возвращать его в нужную точку модели. Такой механизм реализован с помощью рандеву-канала для передачи управления и оператора *goto* для перехода в соответствующее место в модели после обработки исключительной ситуации.

Для удобства моделирования часто используемый код выносится в отдельные *inline*-блоки[9] с «говорящими» именами, и часто такой код

[8]URL: http://spinroot.com/spin/Man/end.html
[9]http://spinroot.com/spin/Man/inline.html

моделирует некоторые функции исходной программы целиком, например, функции создания, слияния и завершения потоков, которые могут вызываться в исполняемых скриптах.

В процессе моделирования естественным образом появлялись переменные и некоторые «процедуры», характерные только для модели. Соответственно, любые действия, связанные с ними, неделимые с точки зрения модели, помещались в блоки *atomic*, которые ограничивают количество чередований (*интерливинга*).

В качестве подробного примера приведем моделирование мьютексов в рамках данной работы (удалена сопроводительная документация):

```
#define N 256
#define mutex bit
#define MutexCount 2
#define _mResetMutex 0
#define _mThreadsMutex 1

mutex mResetMutex = 1;
mutex mThreadsMutex = 1;
typedef mutexes
{
  bool forThread[N] = true;
};
mutexes mutexInfo[MutexCount];

inline lock(_s, __s, _thread)
{
  atomic {
    assert(mutexInfo[__s].forThread[_thread]);
    _s == 1 ->
    mutexInfo[__s].forThread[_thread] = 0;
    _s--;
  };
}
inline unlock(_s, __s, _thread)
{
  atomic {
    assert(_s == 0);
    assert(!mutexInfo[__s].forThread[_thread]);
    mutexInfo[__s].forThread[_thread] = 1;
    _s++;
  };
}
```

```
// Использование:
lock(mThreadsMutex, _mThreadsMutex, _pid);
unlock(mThreadsMutex, _mThreadsMutex, _pid);
```

На данном этапе работы мы абстрагировались в моделируемой компоненте от потоков, отвечающих за работу сенсоров и соединений. Проанализировав, каким образом могли повлиять такие потоки на остальные, мы добавили в модель процесс *User*, который контролирует запуск и остановку исполнения скриптов посредством посылки сообщений-сигналов в очередь событий основного потока приложения. Остальные потоки (см. рис. 2) были смоделированы соответствующими отдельными процессами. Возникающее из-за конечности модели на языке PROMELA ограничение на максимальное количество потоков, создаваемых в процессе исполнения скрипта, является приемлемым в силу того, что исходная программа работает на роботах, где редко требуется огромное количество (более 255) одновременно запущенных потоков.

В процессе моделирования были повсеместно использованы различные приемы абстракции:

- абстракция предикатов,
- абстракция типов данных,
- абстракция наблюдаемых переменных,
- абстракция от вызовов функций и системных вызовов,
- абстракция от «лишнего кода».

Таким образом, число возможных состояний моделируемой системы было существенно уменьшено, что позволило произвести полную проверку пространства состояний.

Стоит отметить, что побочным результатом моделирования стали шесть обнаруженных недочетов в программе, которые явно проявились именно при построении модели. Некоторые из них связаны с недокументированной функциональностью, некоторые — с нежелательными особенностями поведения исходной программы.

4. Требования и спецификации к системе

В связи с отсутствием четких требований для всей программы, для моделируемой компоненты были сформулированы требования на естественном языке, описывающие ожидаемое поведение (свойства) исходной программы.

- Завершение работы скрипта с ошибкой должно корректно обрабатываться: необходимые сообщения об ошибке отображаться на дисплее робота, а система должна корректно завершить все запущенные потоки.

- Прерывание выполнения скрипта любым способом (при нажатии кнопки на роботе и удаленно) должно всегда возвращать систему в исходное состояние, готовое к запуску новых скриптов.

- Запуск выполнения скрипта во время работы другого скрипта должен корректно завершать работу работающего и запускать новый скрипт без побочных эффектов.

- Содержание исполняемых пользовательских скриптов не должно приводить к нежелательному поведению системы. То есть различные запуски скриптов не должны влиять друг на друга, а само исполнение не должно приводить к «падениям» программы, зависаниям и т.д.

Естественным образом к исходной программе были предъявлены общие требования, характерные для любой многопоточной программы, такие как отсутствие гонок, тупиков (некорректных конечных состояний) и активных блокировок.

Описанные выше требования были заданы при помощи операторов `assert` и различных LTL-формул. Опишем ниже заданные LTL-формулы, опустив описание `assert`-ов.

Введем обозначения.

```
// Флаг, который имеет значение true, если сообщение об ошибке не пусто
#define p1 mErrorMessage
```

```
// Метка, обозначающая вывод сообщения об ошибке на дисплей робота
#define q1 GUIThread@showError

// С помощью переменной mState контролируется остановка исполнения скрипта
#define p2 mState != starting

// Флаг, используемый для организации корректного завершения многопоточного исполнения
#define r3 mResetStarted
// Флаг mResetStarted в функции reset принимает значение true
#define p3 GUIThread@mResetStarted_enter
// Флаг mResetStarted в функции reset принимает значение false
#define q3 GUIThread@mResetStarted_exit

// Метка, обозначающая вызов метода запуска скрипта
#define p4 scriptWorkerThread@doRunInvoked
// Метка, обозначающая посылку сигнала о завершении работы скрипта
#define q4 scriptWorkerThread@completedEmitted
// Флаг, который принимает значение true, если поток управления ожидаемо попал в цикл
// и принимает значение false, если поток управления вышел из цикла
#define r4 isAutonomousCycle
```

Зададим LTL-формулы.

$\square(p1 \Rightarrow \lozenge q1)$ — «всегда, если произошла ошибка при исполнении скрипта, соответствующее сообщение выведется на дисплей робота»

$\square\lozenge p2$ — «всегда в ходе нескольких запусков скриптов переменная `mState` не может находиться в одном и том же состоянии `starting` (иначе появится активная блокировка)»

$\square(p3 \Rightarrow (r3 \,\mathsf{U}\, q3))$ — «всегда в промежутке между двумя точками программы флаг `mResetStarted` не должен принимать значение `false` (иначе могут происходить непредвиденные вычисления), а поток управления, пройдя через первую точку, должен обязательно дойти до второй точки»

$\square(p4 \Rightarrow (\lozenge q4) \vee (\lozenge\square r4))$ — «всегда, если был вызван запуск скрипта, должен по итогу быть испущен соответствующий сигнал, сигнализирующий о его завершении, или поток управления должен попасть в цикл бесконечного ожидания (такой вариант допускается для исходной программы)»

При составлении формул полезными оказались шаблоны, предлагаемые [21] и [5].

5. Верификация и анализ контрпримеров

Контрпримеры анализировались и проверялись на реальной системе. В случае, если модель была неадекватной, она модифицировалась. Если проверяемое свойство нарушалось на исходной системе, ошибка записывалась в систему отслеживания ошибок. Было решено записывать ошибки в следующем формате: текущая модель, контрпример с комментариями, а также примеры скриптов, на которых может воспроизвестись ошибка. Таким образом, было найдено тринадцать ошибок, больше половины являются критическими, и некоторые из которых являются активными блокировками. Также было найдено одно предупреждение, которое не было предусмотрено в исходной программе, но отлавливалось на уровне инструментария Qt.

Стоит отметить, что при проверке LTL-формул мы ограничивались только теми вычислениями, для которых выполнялось требование слабой справедливости ввиду того, что окружение времени выполнения работает на операционной системе со справедливым планировщиком.

Опишем подробно ошибки, найденные верификатором.

Может произойти вычисление, на котором (в случае непредвиденной ошибки в ходе исполнения скрипта и посланном до этого сигнале завершения скрипта) может произойти вызов метода `unlock` для одного из мьютексов, для которого не был вызван метод `lock`, что, как сказано в документации инструментария Qt, может привести к неопределенному поведению программы, в частности, ее «падению».

Верификатором было найдено необрабатываемое исключение.

Одна из активных блокировок может быть получена следующим образом: в скрипте создаются два потока, в каждом потоке вызывается метод слияния с другим потоком (то есть два потока будут ожидать завершения работы друг друга), а после этого пользователь пытается завершить исполнения скрипта. При этом происходит активная блокировка основного потока, так как не может произойти корректное завершение работы скрипта.

Следующая обнаруженная ошибка связана с известной особенно-

стью работы скриптового интерпретатора инструментария Qt, а именно «падением» программы, при попытке остановки исполнения скриптового интерпретатора до начала выполнения скрипта [15]. Был поставлен соответствующий `assert`, который и сработал на некотором вычислении. Если вызвать завершение некоторого потока после его создания, то может получиться так, что мы останавливаем скриптовый интерпретатор, который в следующий момент времени попробует исполнить содержимое скрипта, что и приведет к «падению» программы.

Одна из найденных ошибок связана с ситуацией, когда соединение определенного сигнала со слотом происходит после того, как сигнал был послан в ходе работы программы, а это приводит к нежелательному поведению системы. Проверка осуществлялась с помощью специального `atomic`-блока `checkUnhandledSignals`, проверяющего очередь сообщений для указанного потока на отсутствие указанного сигнала-сообщения.

Существует ошибка, найденная верификатором, которая может проявиться при перезапуске скрипта, содержащего смоделированную функцию `quit`. При этом, управление одного из запущенных потоков может попасть в бесконечный цикл, и не завершаться вплоть до выключения робота, расходуя память и процессорное время впустую. По сути, это еще одна активная блокировка.

Необычную ошибку, связанную со временем, удалось обнаружить после моделирования метода `wait`. Флаг для таймера был установлен в положение `false`, которое никак не могло измениться в модели. Объяснение для такого моделирования было простое — время ожидания в передаваемом параметре может быть указано достаточно длительным, чтобы в модели произошло все возможные вычисления. Выйти из цикла ожидания можно было, лишь получив сигнал о завершении исполнения скрипта. В такой модели верификатор нашел некорректное конечное состояние в точке, где и «крутился» цикл, ожидая сигнала о завершении скрипта. Но дело в том, что такой сигнал испускался в соответствующем контрпримере, то есть, в реальной системе получившаяся ситуация интерпретируется следующим образом. Может полу-

читься так, что, если исполняется скрипт, в котором вызывается метод `wait`, и пользователь хочет завершить текущее исполнение или запустить другой скрипт, возможна ситуация, при которой после нажатия кнопки завершения скрипт не закончит свое исполнение сразу, а будет ожидать некоторое время, которое было указано в параметре вызванного метода `wait`, что является нежелательным поведением системы. Кроме этого, пользователю предоставится возможность запускать новые скрипты, несмотря на то, что исполнение последнего скрипта не было закончено. Причина ошибки, скорее всего, кроется в соединении соответствующих сигнала и слота после посылки сигнала. Дальнейший анализ на реальной системе показал, что при воспроизведении такого вычисления скрипт по каким-то причинам исполняется до своего конца, а не просто выходит по таймеру, что делает найденную ошибку еще более актуальной.

Ошибка, связанная с зависанием основного потока приложения (ещё одна активная блокировка), может быть получена следующим образом. Необходимо запустить скрипт, содержащий вызов смоделированного метода `run`, и сразу же послать сигнал завершения скрипта. Тогда может получиться так, что запущенный поток, в котором исполнялся скрипт, «повиснет» в бесконечном цикле ожидания завершения скрипта, и в то же время блокируется основной поток, в котором должны посылаться соответствующие сигналы.

Естественным требованием для реальной системы было отсутствие влияния запусков одних скриптов на запуски других. И для таких случаев верификатор нашел две ошибки. Обе ошибки нашлись с помощью соответствующего `assert`, проверяющего находится ли на момент запуска нового скрипта флаг `mInEventDrivenMode` в начальном состоянии (`false`). В одном случае при любых запусках скрипта, содержащего два вызова смоделированного метода `quit`, флаг не возвращался в начальное состояние и нежелательным образом влиял на запуск следующего скрипта. Вторая ошибка с тем же влиянием возникает не всегда, но может проявиться при наличии в первом скрипте вызовов метода `run` и непредвиденной ошибки после.

Следующая ошибка обнаружилась при проверке свойства, всегда ли сообщение об ошибке (например, непредвиденной) выведется на дисплей робота. Верификатор выдал контрпример, при котором сообщение об ошибке «теряется» в некоторых случаях, и пользователь его не видит, и такое вычисление оказалось возможным на реальной системе.

Была смоделирована переопределенная функция `print`, в реализации которой дополнительно вызывался метод исполнения программы скриптовым интерпретатором. Соответственно, был добавлен `assert`, проверяющий, что отсутствуют вычисления, при которых будет вызвана остановка работы скриптового интерпретатора до запуска исполнения программы. И верификатор нашел контрпример, на котором такой `assert` срабатывает.

Последняя найденная ошибка связана с вызовом метода `unlock` для одного из мьютексов, для которого не был вызван метод `lock`. Ошибка возникает в очень редких случаях, когда в методе `tryLockReset` после верного вызова `unlock` останавливается поток управления, а в другом потоке меняется флаг `mResetStarted`, который является возвращаемым значением для метода `tryLockReset`. И после этого идет проверка изменившегося значения, а после — ошибочный вызов метода `unlock`. Такая ошибка похожа на описанную в начале, тоже связанную с методом `unlock`, но имеет другую природу.

Что касается одного найденного предупреждения, верификатор выдал контрпример, в котором в скрипте пытался вызвать метод слияния потоков, но только с собой. Для верификатора это является некорректным конечным состоянием, однако в реальной системе ошибки нет, выводится следующее сообщение: «QThread::wait: Thread tried to wait on itself», которое не может увидеть пользователь робота, так как такая ошибка отлавливается на уровне инструментария Qt.

Таким образом, формальная верификация позволила обнаружить около десяти критических ошибок в исходной программе окружения времени выполнения, что позволяет говорить об адекватности и корректности построенной в рамках работы документированной модели.

Как уже было сказано, для верификации использовались внутрен-

ние проверки `assert` и LTL-формулы. Было доказано лишь два простых свойства корректности исполнения пустого скрипта и достижения конечного состояния. Большое количество выданных верификатором SPIN контрпримеров затруднило дальнейшее доказательство более существенных свойств системы.

6. Заключение

В данной работе были достигнуты следующие результаты.

- Построена документированная, корректная и адекватная модель верифицируемой системы на языке PROMELA.
- Сформулированы требования на естественном языке и разработаны спецификации к системе на языке LTL на основе ожидаемого поведения исходной системы.
- Проведена верификация с помощью инструмента SPIN, найдены тринадцать ошибок и одно предупреждение.
- В качестве побочного результата при построении модели были найдены шесть недочетов, не связанных с процессом формальной верификации.

Исходный код модели, трассы контрпримеров и комментарии к ним выложены в открытый доступ [10].

Смоделированные компоненты и примитивы инструментария Qt могут быть использованы в других моделях. Модель документирована и может эволюционировать вместе с разработкой исходной программы окружения времени выполнения. Ввиду непредвиденно большого объема работы по анализу контрпримеров, не были доказаны некоторые существенные свойства системы. В качестве возможного продолжения работы существует две идеи:

- продолжить формальное доказательство свойств системы для текущей модели;
- в процессе моделирования большинство системных вызов не были учтены, к примеру, в модели практически полностью игнорируется взаимодействие окружения с драйверами датчиков робота; применение активно развивающегося подхода software model checking видится перспективным для решения этой проблемы.

Список литературы

[1] Bandera: Extracting finite-state models from Java source code / James C Corbett, Matthew B Dwyer, John Hatcliff et al. // Software Engineering, 2000. Proceedings of the 2000 International Conference on / IEEE. — 2000. — P. 439–448.

[2] Holzmann Gerard J. The model checker SPIN // IEEE Transactions on software engineering. — 1997. — Vol. 23, no. 5. — P. 279.

[3] Holzmann Gerard J, H Smith Margaret. Software model checking: extracting verification models from source code† // Software Testing, Verification and Reliability. — 2001. — Vol. 11, no. 2. — P. 65–79.

[4] Johnson Leslie A. DO-178B, Software considerations in airborne systems and equipment certification // Crosstalk, October. — 1998.

[5] LTL-patterns. — 2016. — May. — URL: `http://patterns.projects.cis.ksu.edu/documentation/patterns.shtml`.

[6] Larsen Kim G, Pettersson Paul, Yi Wang. Model-checking for real-time systems // Fundamentals of computation theory / Springer. — 1995. — P. 62–88.

[7] Nusmv 2: An opensource tool for symbolic model checking / Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia et al. // Computer Aided Verification / Springer. — 2002. — P. 359–364.

[8] Peng Hong, Tahar Sofiene, Khendek Ferhat. Comparison of SPIN and VIS for protocol verification // International Journal on Software Tools for Technology Transfer. — 2003. — Vol. 4, no. 2. — P. 234–245.

[9] Pnueli Amir. The temporal logic of programs // Foundations of Computer Science, 1977., 18th Annual Symposium on / IEEE. — 1977. — P. 46–57.

[10] Results. — 2016. — May. — URL: `https://github.com/Kirill-NiK/SpinTrikRuntimeModel`.

[11] Schlich Bastian, Kowalewski Stefan. Model checking C source code for embedded systems // International journal on software tools for technology transfer. — 2009. — Vol. 11, no. 3. — P. 187–202.

[12] Systems and software verification: model-checking techniques and tools / Béatrice Bérard, Michel Bidoit, Alain Finkel et al. — Springer Science & Business Media, 2013.

[13] TRIK. TRIK official site. — 2016. — February. — URL: `http://www.trikset.com/`.

[14] TRIK. trikRuntime architecture. — 2016. — May. — URL: `https://github.com/trikset/trikRuntime/wiki/Архитектура`.

[15] TRIK. trikRuntime running process. — 2016. — May. — URL: `https://github.com/trikset/trikRuntime/wiki/Исполнение-QtScript-на-роботе`.

[16] TRIK. trikRuntime source code. — 2016. — February. — URL: `https://github.com/trikset/trikRuntime`.

[17] VIS: A system for verification and synthesis / Robert K Brayton, Gary D Hachtel, Alberto Sangiovanni-Vincentelli et al. // Computer Aided Verification / Springer. — 1996. — P. 428–432.

[18] Yovine Sergio. Kronos: A verification tool for real-time systems // International Journal on Software Tools for Technology Transfer (STTT). — 1997. — Vol. 1, no. 1. — P. 123–133.

[19] The nuXmv symbolic model checker / Roberto Cavada, Alessandro Cimatti, Michele Dorigatti et al. // Computer Aided Verification / Springer. — 2014. — P. 334–342.

[20] Беляев Алексей Борисович. Верификация алгоритма поддержки транзакционной памяти // Научно-технические ведомости, sТелематика-2010: телекоммуникации, вебтехнологии, суперкомпьютингї. СПбГУ, Санкт-Петербург ќ. — 2010. — Vol. 101. — P. 186–192.

[21] Карпов Ю. Г. Model Checking. Верификация параллельных и распределенных программных систем. — СПб.: БХВ-Петербург, 2010.

[22] Копытов Дмитрий Сергеевич. — 2016. — February. — URL: http://se.math.spbu.ru/SE/diploma/2015/bse/Kopytov_Dmitrij_Sergeevich-text.pdf.

[23] Савенков Константин. Курс лекций «Верификация программ на моделях». — 2016. — April. — URL: http://savenkov.lvk.cs.msu.su/mc.html.

[24] Юрьевич Трифанов Виталий. — 2016. — April. — URL: http://se.math.spbu.ru/SE/diploma/2009/Trifanov.7z.